Towards a Compositional SPIN

Corina S. Păsăreanu and Dimitra Giannakopoulou
{pcorina,dimitra}@email.arc.nasa.gov

QSS and RIACS, NASA Ames Research Center, Moffett Field, CA 94035, USA

Abstract. This paper discusses our initial experience with introducing automated assume-guarantee verification based on learning in the SPIN tool. We believe that compositional verification techniques such as assume-guarantee reasoning could complement the state-reduction techniques that SPIN already supports, thus increasing the size of systems that SPIN can handle. We present a "light-weight" approach to evaluating the benefits of learning-based assume-guarantee reasoning in the context of SPIN: we turn our previous implementation of learning into a main program that externally invokes SPIN to provide the model checking-related answers. Despite its performance overheads (which mandate a future implementation within SPIN itself), this approach provides accurate information about the savings in memory. We have experimented with several versions of learning-based assume guarantee reasoning, including a novel heuristic introduced here for generating component assumptions when their environment is unavailable. We illustrate the benefits of learning-based assume-guarantee reasoning in SPIN through the example of a resource arbiter for a spacecraft.
Keywords: assume-guarantee reasoning, model checking, learning

1 Introduction

This paper describes work performed in the context of a NASA project called Reliable Software Systems Development. The aim of the project is to improve the reliability and safety of software systems to support human and robotic exploration of space. The emphasis is on tool support for the development of verifiable software - tools will be applicable at all stages of the software development, and will target the C language for implementation. For design, the tool that will be supported is SPIN [18] for the following two main reasons. SPIN has been used extensively and successfully for industrial applications. Moreover, SPIN enables embedding of C code, which allows to combine designs with implementations. The users of the tool are thus offered the convenience of using a single environment for verification when transitioning between different phases of the software development.

We present here a component of this project which aims at investigating whether/how compositional techniques can benefit SPIN in dealing with software designs. The compositional techniques that we investigate are based on automated assumption generation for assume-guarantee reasoning, as presented

in [4, 9, 13]. The techniques were implemented in the LTSA tool [12] for the analysis of design models encoded as *finite state* labelled transition systems with blocking communication. Although these techniques have proven effective in the LTSA [25], there is no guarantee that they will be (as) successful in the context of other model checkers. For example, as seen in [14], the savings obtained with automated assume-guarantee reasoning at the design level with LTSA were more pronounced than those obtained at the (Java) code level with the Java PathFinder model checker [30]. The LTSA is by nature a compositional tool, which means that any component in isolation can be targeted for analysis, without the need to provide an environment to turn it into a "closed" system, which is the case for a Java component. Moreover, the amount of detail at the code level makes state spaces larger and may "hide" the size of the savings obtained from a particular approach. SPIN lies somewhere in between the two tools: SPIN's input language – Promela – is closer to a programming language than the input language of the LTSA, but it is still a modeling language, which allows to abstract away implementation details that may hamper verification.

The work reported here is a first study of the issues and benefits of introducing compositional techniques into SPIN (which is not inherently a compositional tool). Our approach has been to make such an evaluation in a "light-weight" fashion, that is, to avoid re-implementing our algorithms within SPIN itself. We will describe how we turned our existing implementations into a main program that invokes SPIN to provide answers to specific model checking questions. As will be discussed later, such an approach has a number of disadvantages, as for example high time overheads. However, we claim that it provides a good way for researchers to make a quick evaluation of the potential benefits of compositional techniques in their model-checking environment. After all, the main interest in model checking is to obtain savings in memory, and these can be evaluated accurately with the framework that we propose.

We will discuss the technical details involved in the implementation of our "light-weight" compositional framework for SPIN. For simplicity, we only look into Promela programs where components communicate in a "rendez-vous" fashion (i.e., Promela channels of size 0). Our evaluations also include a novel heuristic presented in this paper for generating component interface specifications using learning. The description of our approach is given in terms of a running example of a client-server system. We then discuss the application of our techniques to the larger case study of a resource arbiter for a spacecraft, where learning-based assume-guarantee reasoning achieved significant memory gains.

To summarize, the contributions of this paper are: 1) an approach for fast and easy evaluation of the benefits that compositional verification techniques based on learning can bring in the context of any model checker, 2) a description of the technical details involved in the implementation of this approach using SPIN, 3) the discussion of a novel heuristic for learning assumptions of components in isolation, and 4) the application of our approach to a realistic resource arbiter for a spacecraft, for which it achieved significant memory gains over traditional monolithic (non-compositional) model checking.

The remainder of the paper is organized as follows. We give background on assume-guarantee reasoning and learning in Section 2. A description of our proposed approach is provided in Section 3, with the technical details of its implementation using SPIN presented in Section 4. Section 5 discusses our experience with applying our approach to the resource arbiter case study. Finally, Section 6 presents related work and Section 7 concludes the paper.

2 Background

2.1 Assume Guarantee Reasoning

We address the problem of checking *design models* expressed as finite state labeled transition systems. We use compositional techniques for increased scalability. For simplicity, let us consider two software components M_1 and M_2 and a *safety* property P (expressed as a finite state automaton). Reasoning about more than two components will be discussed later in Section 3.

The goal is to check if the two components operate correctly together to achieve the desired property, i.e. to check $M_1||M_2 \models P$ using model checking techniques. Here, the parallel composition operator $||$ denotes the product construction for finite state automata, where the behavior of two components is combined by synchronization of common actions and interleaving of remaining actions. Property P encodes the desired *interactions* between components. Checking $M_1||M_2 \models P$ directly may be too expensive (there may not be enough time and memory resources to complete the computation), so we break-up the verification into two smaller sub-problems, i.e. we check M_1 and M_2 separately, using *assume-guarantee reasoning*.

In the assume-guarantee paradigm a formula is a triple $\langle A \rangle\ M\ \langle P \rangle$, where M is a component, P is a property, and A is an assumption about M's environment. The formula is true if whenever M is part of a system satisfying A, then the system must also guarantee P.

The simplest assume-guarantee proof rule shows that if $\langle A \rangle\ M_1\ \langle P \rangle$ and $\langle true \rangle\ M_2\ \langle A \rangle$ hold, then $\langle true \rangle\ M_1 \parallel M_2\ \langle P \rangle$ also holds. This proof strategy can also be expressed as an inference rule as follows:

$$\frac{\begin{array}{l}\text{(Premise 1) } \langle A \rangle\ M_1\ \langle P \rangle \\ \text{(Premise 2) } \langle true \rangle\ M_2\ \langle A \rangle\end{array}}{\langle true \rangle\ M_1 \parallel M_2\ \langle P \rangle}$$

Thus, using this rule we can show that P holds on $M_1 \parallel M_2$, by checking $\langle A \rangle\ M_1\ \langle P \rangle$ and $\langle true \rangle\ M_2\ \langle A \rangle$ separately. More elaborate rules can be used for this style of reasoning [4]. The underlying aim for all such rules is to make model checking of their premises cheaper, in terms of time and in particular consumed memory, than non-compositional verification. To achieve this, the assumptions have to be *much smaller* than the analyzed components. Coming up with appropriate assumptions is traditionally a difficult, manual process.

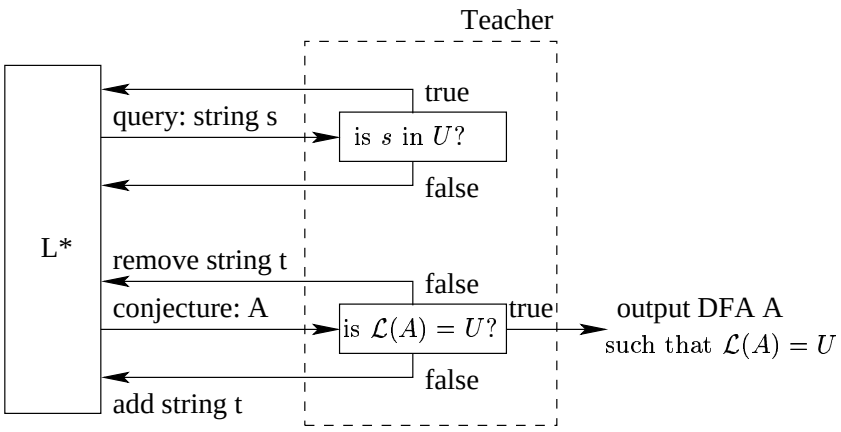

Fig. 1. The L* learning algorithm

In previous work we proposed to use an off-the-shelf learning algorithm, L*, to derive appropriate assumptions *automatically*. Initial approximate assumptions are gradually refined by means of learning from counterexample traces obtained by model checking assume guarantee triples.

2.2 The L* Learning Algorithm

The learning algorithm used by our approach was developed by Angluin and later improved by Rivest and Schapire. We refer to the *improved* version by the name of the original algorithm, L*. L* learns an unknown regular language and produces a DFA that accepts it – see Figure 1. Let U be an unknown regular language over some alphabet Σ. In order to learn U, L* needs to interact with a *Minimally Adequate Teacher*. The Teacher must be able to correctly answer two types of questions from L*. The first type is a *membership query*, consisting of a string $s \in \Sigma^*$; the answer is *true* if $s \in U$, and *false* otherwise. For the second type of question, the learning algorithm generates a *conjecture*, i.e., a candidate DFA A whose language the algorithm believes to be identical to U. The answer is *true* if $\mathcal{L}(A) = U$. Otherwise the Teacher returns a counterexample, which is a string s in the symmetric difference of $\mathcal{L}(A)$ and U.

At a higher level, L* creates a table where it incrementally records whether strings in Σ^* belong to U. It does this by making membership queries to the teacher. At various stages L* decides to make a conjecture. It constructs a candidate automaton A based on the information contained in the table and asks the Teacher whether the conjecture is correct. If it is, the algorithm terminates. Otherwise, L* uses the counterexample returned by the Teacher to extend the table with strings that witness differences between $\mathcal{L}(A)$ and U.

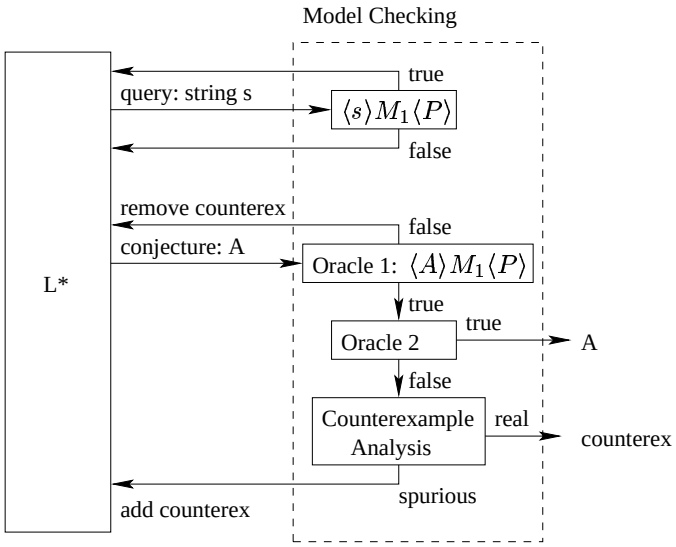

Fig. 2. Tool Architecture

Characteristics of L*. L* is guaranteed to terminate with a minimal automaton A for the unknown language U. The conjectures made by L* strictly increase in size; each conjecture is smaller than the next one, and all incorrect conjectures are smaller than A. Therefore, if A has n states, L* makes at most $n-1$ incorrect conjectures. The number of membership queries made by L* is $\mathcal{O}\left(kn^2 + n\log m\right)$, where k is the size of the alphabet of U, n is the number of states in the minimal DFA for U, and m is the length of the longest counterexample returned when a conjecture is made.

3 Tool Architecture

We present here an initial study for a tool-based approach to compositional verification, that uses the L* algorithm to build assumptions and the SPIN model checking tool to check assume guarantee triples. Although using learning to automate assume guarantee reasoning was introduced in our previous work, there are some novel ideas that we propose here:

- We present a *generic* tool architecture that uses learning for automated assume guarantee reasoning for multiple components. By generic, we mean that the tool can be instantiated with different model checking tools for checking assume guarantee triples; we discuss the use of SPIN here.
- The tool can be used for checking different assume guarantee rules (as before), but in addition we present a novel heuristic that allows us to derive the *interface specification* for a component M_1, in the absence of a specification

of its environment (i.e. M_2). This interface specification can be used to check if the component M_1 behaves correctly in multiple contexts. In the past, we have experimented with an approach that uses conformance checking [9]. Instead of using this expensive approach, we present a light-weight heuristic that enables cheaper generation of precise interface specifications.

The architecture of the compositional verification tool is illustrated in Figure 2. The architecture is derived from our previous work on compositional verification [9]. The goal is to use learning to derive an assumption A such that the assume guarantee triple $\langle A \rangle\ M_1\ \langle P \rangle$ evaluates to *true*. The *weakest assumption* A_w under which M_1 satisfies P is such that, for any environment component E, $\langle true \rangle\ M_1 \parallel E\ \langle P \rangle$ if and only if $\langle true \rangle\ E\ \langle A_w \rangle$. In our framework, L* attempts to build A_w through iterative learning. For L* to learn A_w, we need to provide a Teacher that is able to answer the two different kinds of questions that L* asks. Our approach uses model checking to implement such a Teacher.

Membership Queries To answer a membership query for s the Teacher simulates s to check if it may lead to a violation. For simplicity, our current implementation for SPIN reduces the simulation to model checking $\langle s \rangle\ M_1\ \langle P \rangle$ (here we abuse the notation and we let s denote also the automaton that accepts string s). If there is no violation, it means that $s \in \mathcal{L}(A_w)$, because M_1 does not violate P in the context of s, so the Teacher returns true. Otherwise, the answer to the membership query is false.

Conjectures Our framework uses the conjectures returned by L* as intermediate candidate assumptions A. The teacher uses two *oracles*: *Oracle 1* guides L* towards a conjecture that is strong enough to make $\langle A \rangle\ M_1\ \langle P \rangle$ true. Once this is accomplished, the resulting conjecture may be too strong, in which case our framework uses *Oracle 2* to guide L* towards a *weaker* conjecture. There are many options for implementing *Oracle 2*, and we discuss some of them below.

Oracle 1 checks $\langle A \rangle\ M_1\ \langle P \rangle$. If this does not hold, the model checker returns a counterexample. The Teacher informs L* that its conjecture A is not correct and provides the counterexample to witness this fact. If, instead, $\langle A \rangle\ M_1\ \langle P \rangle$ holds, the Teacher forwards A to Oracle 2.

Oracle 2 needs to ensure that the candidate assumption is indeed the *weakest*. In the context of this work, we have implemented different versions for this oracle.

- If component M_2 is available then the oracle checks $\langle true \rangle\ M_2\ \langle A \rangle$ (as in our previous work). If the result of model checking is true, then, according to the assume-guarantee rule, P holds on $M_1||M_2$. The teacher therefore returns true, whether A represents the weakest assumption or not, because the computed assumption (smaller or equal in size to A_w) is good enough to prove that the property holds. If model checking returns a counterexample, our implementation performs counterexample analysis. If the counterexample indicates a real error, the framework stops and the error is reported to the

user. Otherwise, the counterexample indicates that the candidate assumption needs to be refined and it is returned to guide L*. Our implementation has been extended to reasoning for n components $M_1||\ M_2||\ ...||M_n$. The system is decomposed into two parts M_1 and $M_2' = M_2||...||M_n$ and the learning algorithm is invoked recursively for checking the second premise of the rule.
- We have implemented a different version of *Oracle 2* (described in detail below), which leads to the generation of the *interface specification* of M_1, and it does not use M_2.

3.1 Generation of Interface Specifications

As discussed, Oracle 2 is responsible for ensuring that an assumption A, shown strong enough by Oracle 1, is not too strong. In other words, the assumption should include all traces over the alphabet of the assumption, in the context of which M_1 satisfies the property P. By alphabet we mean the set of events that are involved in a state machine.

We discuss here the case where the alphabet of the property and the alphabet of the assumption are the same. We restrict ourselves to this case for simplicity, but also because it covers all the examples that we discuss in this paper. We are currently studying different cases and plan on extending the proposed heuristic heuristic for those.

Let T_A denote the set of all traces over the alphabet of the assumption A. Then A should include all traces in T_A that satisfy the property; if some trace $t \in T_A$ that satisfies P is not in the current candidate assumption A, then A is too imprecise, so t is returned to the learning algorithm for the assumption to be refined. The above check can be formulated as $P \models A$, and can be performed by a model checker, with the counterexamples returned to the learning algorithm. Our proposed heuristic for Oracle 2 for generating interface specifications is to therefore implement $P \models A$.

Note that our heuristic is not always accurate, meaning that it may fail to report traces that the assumption does not include even though it should. The traces that it may miss are traces that violate P but that will never be exercised in the context of the component M_1. These traces are the traces of $!M_1||!P$, where $!M_1$ denotes the complement of M_1, and similarly for P. Computing the complement of M_1 involves determinization, which may increase the state-space of M_1 exponentially, in the worst case. For this reason, we do not include this check in our heuristic. One may argue that many components do not exhibit this worst-case complexity. For such components, however, rather than computing $!M_1||!P$, it would make more sense to construct the assumption directly, using the algorithm presented in our previous work [13]. Learning was introduced in [9] in order to avoid the potential complexity of the computation presented in [13].

It is worth mentioning that, although our heuristic as currently implemented may not always compute the weakest assumption, our experiments discussed later in the paper demonstrate that it is quite effective in practice.

```
mtype = {u1, u2, Nobody};
chan request = [0] of {mtype};
chan cancel = [0] of {mtype};
chan grant = [0] of {mtype};
chan deny = [0] of {mtype};

active proctype server() {
 mtype resUser = Nobody;
 mtype u;
 S0: if
     :: request?u ->
         if
         :: (resUser == Nobody) -> grant!u; resUser = u; goto S0;
         :: else -> deny!u; goto S0;
         fi;
     :: cancel?u ->
         if
         :: (resUser == u) ->  resUser = Nobody;  goto S0;
         :: else -> goto S0;
         fi;
     fi;
}
```

Fig. 3. Promela code for server

4 Implementation

Our implementation makes use of our previous Java implementation of L* in the context of the LTSA tool [9, 12]. The implementation supports the analysis of multiple components through recursive invocation and the new heuristic for Oracle 2. Moreover, the learning now runs as a stand-alone application that invokes SPIN (from within Java) to answer queries and conjectures.

We consider here only a subset of Promela, where components are Promela processes that communicate through rendezvous channels. We consider safety properties that refer to the rendezvous communication between components. We leave for future work the extension of the approach to handling the full Promela language. We selected this subset of Promela because it bears a close correspondence to the type of models that we analyze in the context of LTSA. Moreover, several systems can be described in this subset. For example, the work presented in [10, 27] shows how abstracted Java and Ada programs can be translated into this exact subset of Promela.

We illustrate the implementation on a simple Promela model for a client server application – see Figure 3 and Figure 4 (left). The model has a *server* and two *clients* that communicate through global rendezvous channels. Note that the MER case study is a more complex version of this type of system.

```
proctype client (mtype u) {
 Init:   if
         :: request!u
         fi;
 PendingReservation:
         if
         ::grant?eval(u)
         ::deny?eval(u) -> goto Init;
         fi;
 PendingCancel:
         if
         :: cancel!u -> goto Init
         fi;
}
```

```
trace {
Q0: if
    ::grant?u2 -> goto Q4;
    ::grant?u1 -> goto Q5;
    fi;
Q4: if
    ::cancel?u2 -> goto Q0;
    fi;
Q5: if
    ::cancel?u1 -> goto Q0;
    fi;
}
```

Fig. 4. Promela code for client (left) and mutual exclusion property (right)

The clients send *requests* to make a reservation for using a common resource, they wait for the server to *grant* the reservation, they use the resource, after which they *cancel* the reservation. The server can *grant* or *deny* a *request*, such that the resource is used only by one client at a time. We analyzed a property stating that the resource shall be used mutually exclusive.

There are many ways of encoding (safety) properties in SPIN: i.e. as basic assertions, never claims or trace assertions [19]. We chose to encode properties as *trace assertions*: the types of safety properties that we typically encounter refer to valid sequences of channel operations, and trace assertions are specifically designed for formulating such sequences. In Section 5 we discuss other formalisms for encoding assume guarantee triples. Figure 4 (right) shows the trace assertion for the mutual exclusion property. The assertion specifies the correctness requirement that receive operations on channel `grant` with `u1` and `u2` alternate with receive operations on `cancel` with `u1` and `u2`, respectively. In other words, for mutual exclusion to be guaranteed, when a user is granted the resource, then this user needs to cancel it before it gets granted to a different user. The trace assertion defines an automaton that monitors the system execution (it changes state when a channel operation that is within its scope is executed).

In order to analyze this model using our learning based implementation, we first brake up the system into its components, i.e. processes `client(u1)`, `client(u2)` and `server()`. We also need to provide the *alphabet* of actions for the candidate assumptions. As discussed in Section 3.1, we set the alphabet of the assumptions to be the same as the alphabet of properties.

Checking Assume Guarantee Triples In our approach, we use SPIN to answer queries and oracles, which are encoded as assume guarantee triples of the form $\langle A \rangle\ M\ \langle P \rangle$. Here A denotes a deterministic finite state automaton that may encode traces (in the case of queries) or candidate assumptions generated by L*. Property P is also a deterministic finite state automaton (encoded as a trace

```
active proctype query () {
 grant!u1;
 grant!u2;
}

active proctype UniversalEnv() {
 do /* actions unmatched in U1||A */
 :: request?u1
 :: deny!u1

 /* actions of other users */
 :: grant?u2
 :: cancel!u2
 :: grant?u3
 :: cancel!u3
 :: grant?u4
 :: cancel!u4
 :: grant?u5
 :: cancel!u5
 od
}
```

```
active proctype CandidateAssumption(){
 Q0:    if
        :: grant!u1-> goto Q2;
        :: grant!u2-> goto Q3;
        :: cancel?u1-> goto Q1;
        fi;
 Q1:    if
        :: grant!u1-> goto Q1;
        :: grant!u2-> goto Q1;
        :: cancel?u1-> goto Q1;
        :: cancel?u2-> goto Q1;
        fi;
 Q2:    if
        :: grant!u1-> goto Q1;
        :: cancel?u1-> goto Q0;
        fi;
 Q3:    if
        :: cancel?u1-> goto Q1;
        :: cancel?u2-> goto Q0;
        fi;
}
```

Fig. 5. Promela code for a query, an assumption and the universal environment

assertion). The assumptions define execution *environments* for the components under analysis. We therefore encode them as Promela processes that run in parallel with the analyzed components (and thus restrict their behavior). The assumption A and the property P are used to examine the component M and to check whether behaviors that are allowed by the assumption may lead to a property violation.

To check an assume guarantee triple, the teacher first creates a file that encodes the assumption as a Promela process and the property as a trace assertion, and it invokes SPIN i.e., it executes the following commands:

```
spin -a M1.promela
cc -o pan pan.c -DSAFETY
./pan -E
```

The teacher waits for the verification to complete and it parses the output of the verification process to check if there were any assertion violations, in which case it returns *false* (together with the counterexample reported by SPIN) to the L* algorithm; otherwise, it returns *true*. All these steps are automated.

As an example, Figure 5 shows the Promela process for checking a `query` on component `client(u1)` for string "`grant!u1; grant!u2;`". Figure 5 also shows the Promela process for a `CandidateAssumption` for `client(u1)`.

We should note that both properties and assumptions are *global*, i.e. they may refer to actions that are not local to the component under analysis. In order to check in isolation whether a component violates a global property, we need to

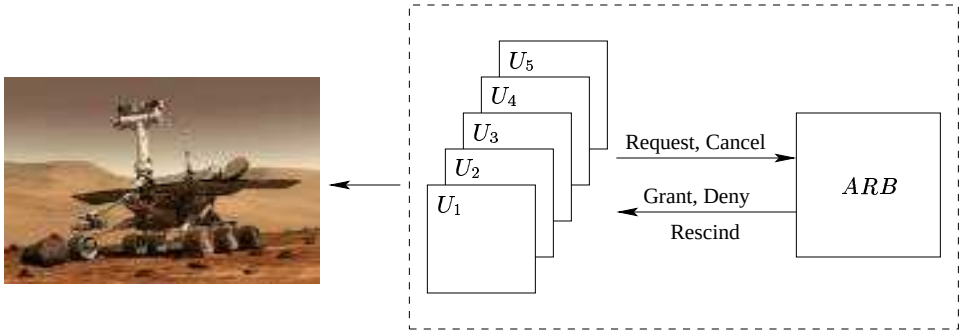

Fig. 6. Arbiter Architecture

provide an environment that substitutes the rest of the system, as typically performed in model checking. In the context of checking assume-guarantee triples, the environment is the universal environment as restricted by the assumption. To simulate that, we provide for each component a universal environment for those rendezvous actions that are not matched with actions in the provided assumption. For example, Figure 5 shows such a closing environment for `client(u1)` – in an infinite loop, the process performs rendezvous for the actions that are unmatched by `client(u1)` and by the process encoding the assumption. Note that the same universal environment is used for checking all the queries and oracles for one particular component (and one property).

5 Analysis of a Resource Arbiter

5.1 Description

We experimented with our approach in the context of a model derived from a component of the flight software for JPL's Mars Exploration Rovers (MER) (see Figure 6). The MER software contains 11 user threads (U_i). Each thread serves one specific application, such as imaging, controlling the robot arm, communicating with earth, and driving. There are 15 shared resources on the rover, to which access is controlled by an arbiter module (ARB). The arbiter module prevents potential conflicts between resource requests, and enforces priorities. For instance, it would not make sense to start a communication session with earth while the rover is driving. The system has been analyzed with SPIN before, in a non-compositional way - a detailed description can be found in [20].

5.2 Analysis

We present here the results of applying compositional analysis for a subproblem with 5 user threads and 5 shared resources. A design-level Promela model of the

Table 1. Arbiter Analysis Results

Analysis	MEM	State Space	Time: $t_{model} + t_{compile} + t_{run}$	Assumption Size
Monolithic	544.019 MB	$3.91653e+06$	$0.021s + 0.854s + 33.745s$	N/A
Recursive	2.622 MB	1002	$0.038s + 1.142s + 0.032s$	6 .. 12
Heuristic	2.622 MB	2941	$0.044s + 1.392s + 0.021s$	12

system was created based on available documentation (3000 lines of Promela code) and was used to check several properties. We report here the results for checking a mutual exclusion property (P) stating that communication and driving can not happen at the same time.

The compositional techniques discussed in this paper work on a specific ordering of the components in the system. For the analyzed system, we ordered the user components first as (U_1 ... U_5) and the arbiter module last as (ARB). As described in [7], compositional techniques tend to be sensitive to different decompositions of a system. The reason we selected this particular ordering was that part of the project involved experimenting with generating assumptions in the absence of an arbiter component.

We then used the learning tool described in Section 3 to generate automatically assumptions A_1 ... A_5 such that:

$$\begin{gathered}\langle A_1\rangle\ U_1\ \langle P\rangle\\ \langle A_2\rangle\ U_2\ \langle A_1\rangle\\ \langle A_3\rangle\ U_3\ \langle A_2\rangle\\ \langle A_4\rangle\ U_4\ \langle A_3\rangle\\ \langle A_5\rangle\ U_5\ \langle A_4\rangle\\ \langle true\rangle\ ARB\ \langle A_5\rangle\end{gathered}$$

For this purpose, we manually created environments that exercise each component, as described in the previous section. We also specified the alphabet of interface actions to be used for building the assumptions. We experimented with the recursive technique that we have implemented for handling multiple components and with the heuristic approach, that analyzes one component at a time. In both cases we were able to compute assumptions for the above premises to hold. Hence, according to the compositional rule presented in Section 2, we concluded that the system $U_1||U_2||U_3||U_4||U_5||ARB$ indeed satisfies P.

The results of the analysis applied to the arbiter system are shown in Table 1. We used a 2.2 GHz dual processor Pentium with 1 Gb of memory running Red Hat Enterprise Linux WS. In the table, row "Monolithic" reports the results obtained from the verification of the system in a non-compositional way, and rows "Recursive" and "Heuristic" report the results obtained by the application of the recursive learning scheme and the heuristic described in Section 3.1, respectively. Specifically, we report the memory and time consumed for verification of the system. For the compositional techniques, the reported time and memory refer to the maximum time or memory consumed to for checking a single premise. They do not include the process of generating the assumptions (reported in

Table 2. Cost of Assumption Generation

Analysis	queries	oracle 1	oracle 2	$t_{SPIN} + t_{Learn}$	MEM_{Learn}	t_{LTSA}	MEM_{LTSA}
Recursive	4884	48	1	5646.365s	1743 K	42.87 s	20400K
Heuristic (A_1)	852	12	1	818.213s	508 K	3.076 s	4845K

Table 2), but rather the process of applying the assume-guarantee premises once the assumptions are available.

The reported times are divided into three parts: t_{model} is the time to create a C model from a Promela model, $t_{compile}$ is the compilation time, and t_{run} is the time to run the specific verification task in SPIN. We also report the size of the assumptions used for compositional verification. Using the recursive algorithm yields assumptions that have 12 states (A_1, A_2 and A_3) and 6 states (A_4 and A_5) while the heuristic approach yields assumptions of size 12 for each component (for this case study, all the assumptions generated using the heuristic approach are the weakest). We need to study further the trade-offs between the two learning approaches: the heuristic approach has the advantage that it can be used for the analysis of a component in isolation (in the absence of the rest of the over-all system, and maybe even before it is available), while the recursive approach may yield smaller assumptions (as it is the case here). This is expected to happen for some systems, because the recursive approach has knowledge of the environment of each component, and may therefore produce stronger (and smaller) assumptions.

The results indicate that compositional verification can achieve significant memory savings over non-compositional verification.

Cost of assumption generation Table 1 reports the results of compositional analysis using assumptions that are already available. Let us now analyze the cost of building these assumptions using learning based techniques. Table 2 reports the results of running the two learning approaches for assumption generation: for the recursive approach, we report the number of queries, the number of oracle invocations and the total time for running the algorithm (this includes t_{Learn} – the time of running the Java implementation that makes external calls to SPIN – plus t_{SPIN} – the total time of running SPIN multiple times for answering queries and conjectures). For interface generation, we report the same data for the generation of an assumption for *only one* component (U_1) – the results for the rest of the components are similar. Therefore, the total time of generating all five interface specifications is approximately 4095 s (5 times 819 s). Table 2 also reports MEM_{Learn} – the memory consumed by our Java implementation (this does not include the memory consumed by a SPIN run – which is reported in Table 1).

Our experiments indicate a serious time overhead, where a dominant factor is the compilation time for queries. For example, there are 852 queries made for the generation of the interface specification of component U_1, and the cost of

```
never {
 do
 :: grant_u1 -> break
 :: !grant_u1 && !grant_u2 && !cancel_u1 && !cancel_u2
 od;
 do
 :: grant_u2 -> break
 :: !grant_u1 && !grant_u2 && !cancel_u1 && !cancel_u2
 od;
 do
 :: !grant_u1 && !grant_u2 && !cancel_u1 && !cancel_u2
 od;
}
```

Fig. 7. A query encoded as a never claim

running a query is $0.045s + 1.283s + 0.011s$, where the compilation time $1.283s$ clearly dominates.

Therefore we looked into ways of reducing the compilation time overhead for queries. In particular, we experimented with an alternative way of encoding queries – as never claims – in order to take advantage of the SPIN's feature that allows for the *separate compilation* of a model and of properties (written as never claims). Note that never claims can be used not only to define correctness properties, but also to *restrict* the search of the verifier to a user-defined subset of the system [19]. It is in the latter fashion that we use never-claims to attempt more efficient checking of queries.

As an example, Figure 7 shows the never claim used for checking a query "grant!u1; grant!u2;" (the analog of the query in Figure 5). Here `grant_u1`, `grant_u2`, `cancel_u1` and `cancel_u2` are global boolean flags added to the Promela model of a component. They are set to true whenever a corresponding rendezvous occurs and are reset to false on any other action. For example, `grant_u1` is set to true (while all the other flags are reset to false) atomically with `grant?u1`. The reason we use these flags is that SPIN does not allow rendezvous actions in never claims. The effect is that the never claim restricts a verification run to all the states that conform to the trace (note that the flags need to be reset after every system step execution, to make sure that the never claim restricts correctly the system). For technical reasons (SPIN does not allow never claims and trace assertions to be checked at the same time), we changed the encoding for properties (as monitors). The encoding of queries as never claims allows us to compile the component model combined with the property only once and to compile separately the never claims for each query. Note that the same approach can be used for encoding assumptions.

With this new encoding, we obtained a significant reduction in running time. For example, the cost of heuristic interface generation for U_1 was reduced by a factor of 4 (from $818.213s$ to $185.185s$). We expect a similar reduction to be

obtained for running the recursive algorithm, and even further reduction for the separate compilation of assumptions.

5.3 Discussion

The implementation described is a first step towards introducing learning-based assume guarantee reasoning in the SPIN model checker. The purpose of this work is fast experimentation with the algorithms in the context of examples encoded in Promela. We intend to explore several directions for improving the performance of this approach in future stages of the project.

The current implementation invokes SPIN for each query and for the two oracles. This involves creating appropriate Promela files, compiling them and running the verification at each step. While this approach works well for small examples, for realistic (large) examples, parsing and compiling the Promela files at each step is costly in terms of time. We believe that a first step towards a better integration will be the creation of specialized algorithms for efficient trace simulation (for checking queries) and for checking properties in the presence of restricting assumptions; these algorithms should allow for separate compilation of models, assumptions and properties.

We should note that we encountered similar timing overheads with the implementation of the learning assume-guarantee approach as a plugin for the LTSA model checker [12], as compared to our initial implementation within the core of the LTSA tool [9]. In that implementation, we encountered a significant performance overhead due to the fact that the plugin communicates with the LTSA by placing descriptions of the models in the Edit tab. As a result, each query or conjecture would require parsing and computing the component model. The avenue we took to solve the problem was to implement our techniques in the core of the LTSA and expose them to the LTSA plugins, while keeping the interfacing for our assume-guarantee reasoning as an LTSA plugin. As a result, the running time of our iterative learning algorithms is low.

For example, the last two columns in Table 2 show the results of running the LTSA implementation for the arbiter case study. The results indicate that an implementation directly in SPIN is likely to similarly improve the performance significantly. Note that part of the gain of having the learning algorithms run within LTSA is that the LTSA can store the results of a particular composition (for a component, for example) and use it in the analysis of multiple properties. The impact can be great in the evaluation of queries, and it may be worth adding this capability in SPIN, for cases where that would be appropriate (when, for example, the component state space is manageable).

A nice feature that the LTSA supports is that the plugin can extend the user interface of the tool, and can be invoked from the LTSA's graphical user interface. As a result, the user can easily customize their assume-guarantee problem, i.e., select the modules and properties that participate in a compositional proof, as well as the rule that is to be applied. In the future, we would like to take a similar approach in integrating our techniques using XSpin. To achieve this, we need to understand better what mechanisms are available or can be added for achieving

Spin/XSpin extensions. Ideally, we would like to display all the components (i.e. processes) in a Promela specification, and to allow the user to choose which components to analyze using assume guarantee reasoning.

6 Related Work

Assume-guarantee reasoning [8, 16, 22, 28] is based on the observation that large systems are being build from components and that this composition can be leveraged to improve the performance of analysis techniques. To reason formally about components in isolation, some form of assumption (either implicit or explicit) about the interaction with, or interference from, the environment has to be made. Several frameworks have been proposed to support this style of reasoning. For example, the Calvin tool [11] provides support for assume guarantee reasoning for the analysis of Java programs, while the Mocha toolkit [2] provides support for modular verification of components with requirement specifications based on the Alternating-time Temporal logic. However, the practical impact of these previous approaches has been limited because they require non-trivial human input in defining appropriate assumptions.

As mentioned, in previous work [9, 13], we have developed techniques for performing assume-guarantee reasoning of software in a *fully automated* fashion. Our techniques target components with message-passing communication - a paradigm used in NASA mission critical software (e.g. MER code). The approach presented in [9] uses L* to build incrementally appropriate assumption, and it forms the basis of the work presented in this paper. Since then, several assume guarantee reasoning frameworks that use L* for learning assumptions have been developed – [3] (see also [1]) presents a symbolic approach to assumption leaning, while [5, 6] use learning based assume guarantee verification for communicating finite state automata specifications extracted from C code. The work presented here is a first attempt to introduce *automated* assume guarantee reasoning in SPIN. In the past [27] we have studied the use of assume guarantee reasoning in the context of SPIN – however, in that work, the assumptions were provided manually by the user.

A related effort [17] includes a framework for thread-modular abstraction refinement, in which assumptions and guarantees are both refined in an iterative fashion. The framework applies to programs that communicate through shared variables, and uses predicate abstraction techniques for the iterative construction of assumptions.

The problem of generating an assumption for a component is similar to the problem of generating component interfaces to deal with intermediate state explosion in compositional reachability analysis. Several approaches have been defined for automatically abstracting a component's environment to obtain interfaces [7, 23]. These approaches do not address the incremental refinement of interfaces.

A number of machine learning approaches has been investigated recently in the context of software verification, with a goal different then ours. One approach

uses learning for computing the set of reachable states in regular model checking [29]. The work in [15] uses the L* to generate a model of a software system in a black box fashion; the model then be fed to a model checker for analysis. Similarly, [21] presents learning techniques for building software models for verification, while a recent approach [24] uses inductive learning to build precise abstractions for program analysis.

7 Conclusions and Future Work

In this paper we discussed our initial experience with automated assume guarantee verification based on learning in the context of SPIN. We presented a light-weight tool that uses learning to build assumptions incrementally and that makes external calls to SPIN to provide all the model checking related answers. We discussed the application of the tool for the verification of a realistic software system – the resource arbiter for a space craft – which resulted in significant memory gains as compared to traditional monolithic model checking.

While this light-weight implementation allows for a quick evaluation of the merits of learning based assume guarantee reasoning in SPIN, it may result in serious performance overheads and we discussed in the paper ways of improving our implementation. In the future, we plan to work towards a tighter integration in SPIN and to investigate how we can further improve the performance of our approach. One possible way is to run in parallel the checks for multiple queries. We also plan to study how our algorithms extend to alternative communication mechanisms (buffered message passing) and to handling liveness properties – the work on learning infinitary regular sets [26] may be a good start in this direction. Another issue that we want to investigate is to make a finer distinction in our algorithms between the interface actions of a component (i.e. to distinguish channel read and write operations) and to study how this affects our approach.

Acknowledgements

The authors would like to thank Gerard Holzmann for his invaluable support and guidance throughout this project.

References

1. R. Alur, P. Cerny, P. Madhusudan, and W. Nam:. Synthesis of interface specifications for java classes. In *Proc. of 32nd POPL*, pages 98–109, 2005.
2. R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proc. of the Tenth Int. Conf. on Comp.-Aided Verification (CAV)*, pages 521–525, June 28–July 2, 1998.
3. R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Proc. of 17th International Conference CAV*, pages 548–562, 2005.

4. H. Barringer, D. Giannakopoulou, and C. S. Păsăreanu. Proof rules for automated compositional verification through learning. In *Int. Workshop on Specification and Verification of Component-Based Sys.*, Sept. 2003.
5. S. Chaki, E. Clarke, D. Giannakopoulou, and C. Pasareanu. Abstraction and assume-guarantee reasoning for automated software verification. Technical report, RIACS, 2004.
6. S. Chaki, E. M. Clarke, N. Sinha, and P. Thati. Automated assume-guarantee reasoning for simulation conformance. In *Proc. of 17th International Conference CAV*, pages 534–547, 2005.
7. S. C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Trans. on Soft. Eng. and Methodology*, 5(4):334–377, Oct. 1996.
8. E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proc. of the Fourth Symp. on Logic in Comp. Sci.*, pages 353–362, June 1989.
9. J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *9th International Conference for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *LNCS*, Warsaw, Poland, 2003. Springer.
10. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. of the* 22^{nd} *Int. Conf. on Soft. Eng.*, June 2000.
11. C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Proc. of the Eleventh European Symp. on Prog.*, pages 262–277, Apr. 2002.
12. D. Giannakopoulou and C. S. Păsăreanu. Learning-based assume-guarantee verification (tool presentation). In *Proc. of SPIN'05 Workshop*, volume 3639 of *Lecture Notes in Computer Science*. Springer, 2005.
13. D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *Proc. of the Seventeenth IEEE Int. Conf. on Auto. Soft. Eng.*, Sept. 2002.
14. D. Giannakopoulou, C. S. Păsăreanu, and J. M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *Int. Conf. on Soft. Eng.*, pages 211–220, May 2004.
15. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Proc. of the Eighth Int. Conf. on Tools and Alg. for the Construction and Analysis of Sys.*, pages 357–370, Apr. 2002.
16. O. Grumberg and D. E. Long. Model checking and modular verification. In *Proc. of the Second Int. Conf. on Concurrency Theory*, pages 250–265, Aug. 1991.
17. T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Proc. of PLDI*, pages 1–13, 2004.
18. G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Soft. Eng.*, 23(5):279–295, May 1997.
19. G. J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Publ., 2003.
20. G. J. Holzmann and R. Joshi. Model driven software verification. In *Proc. of 11th International SPIN Workshop*, pages 76–91, Oct. 2004.
21. H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *Proc. of 15th International Conference CAV*, pages 315–327, 2003.
22. C. B. Jones. Specification and design of (parallel) programs. In R. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332. IFIP: North Holland, 1983.

23. J.-P. Krimm and L. Mounier. Compositional state space generation from Lotos programs. In *Proc. of the Third Int. Workshop on Tools and Alg. for the Construction and Analysis of Sys.*, pages 239–258, Apr. 1997.
24. A. Loginov, T. W. Reps, and S. Sagiv. Abstraction refinement via inductive learning. In *Proc. of 17th International Conference CAV*, pages 519–533, 2005.
25. J. Magee and J. Kramer. *Concurrency: State Models & Java Programs.* John Wiley & Sons, 1999.
26. O. Maler and A. Pnueli. On the Learnability of Infinitary Regular Sets. *Information and Computation*, 118(2), 1995.
27. C. S. Păsăreanu, M. B. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Comp. Sci.*, pages 168–183. Springer-Verlag, Sept. 1999.
28. A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logic and Models of Concurrent Systems*, volume 13, pages 123–144, New York, 1984. Springer-Verlag.
29. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Using language inference to verify omega-regular properties. In *Proc. of 11th International Conference TACAS*, pages 45–60, 2005.
30. W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In *Proc. of the Fifteenth IEEE Int. Conf. on Auto. Soft. Eng.*, pages 3–12, Sept. 2000.